"Vasile Alecsandri" University of Bacău
Faculty of Sciences
Scientific Studies and Research
Series Mathematics and Informatics
Vol. 24 (2014), No. 1, 91-104

FUNCTIONAL FOLD BASED PROGRAMMING IN SWI-PROLOG

DAN POPA

Abstract. In this paper the author is completing a gap in the style used by SWI-Prolog programmers. Important notions and theorems from the field of functional programming can now migrate to the logic programming paradigm: foldl, foldr, the universality property, etc.

1. INTRODUCTION

Processing of lists and containers is usually performed using, especially :

1. loops or iterators (in C respectively C++),

2. recursive functions (C, C++, Pascal, Haskell and functional languages) and recursive predicates (standard dialects of Prolog, other logic programming languages),

3. recursive data types used "à la Prolog" (possible in Haskell due to the Prolog like style of using Hindley-Milner type inference system),

4. foldl and foldr recursion operators are usable in some functional languages (Haskell, ML, metaML, etc).

Keywords and phrases: fold, foldl, foldr, functional programming, logic programming
(2010) Mathematics Subject Classification: 68N17, 68N18

In this paper we are using the techniques from the fourth position of the above list to improve the set of tools available for the SWI-Prolog programmers, by adding the folds and the related theorems into the "backpack" of tools used in logic programming. Also note that this is possible due to some extensions of the standard Prolog, available in SWI-Prolog, which can facilitates crossing the border between functions and predicates, i.e., specially declared, predicates can be used like functions and conversely..

2. DEFINITIONS OF THE FOLDS

2.1. In the functional programming language Haskell, foldr and foldl are the names of two high level functions, (i.e. functions which are using other functions as arguments). There are many ways of introducing folds but a good starting point can be found in [3], where is written in Gofer /Haskell like style.

Definition 1. A fold is a common pattern of recursive processing on lists (but some similar results can be achieved on other types of containers) being defined as:

```
foldr :: (a->b->b) -> b-> [a]->b

foldr op v0 [] = v0

foldr op v0 (head:tail) = op head (foldr op v0 tail)
```

where op :: (a->b->b) is a not necessary associative operator.

Here (and the type/set a is not necessary identical with the type/set b, but can be). The (head:tail) is simply the pattern of the lists, divided in head and tail by the cons operator, noted in Haskell by a column (:).

During the computation, the above operator op is applied backwards, starting from the tail and finishing with the head. There also exists another fold, which is processing lists in reverse order, starting from the head of the

list:

```
foldl :: (a->b->a) -> a -> [b] -> a

foldl op vn [] = vn

foldl op vn (head:tail) = foldl op (op vn head) tail
```

During list processing the above operator op is applied started from the left to the right, starting from the head to the tail.

The both folds operators are interesting because:

1. the recursion pattern is present in a lot of computations and a lot of common used functions are in fact folds. The well known catenation of strings and the map operator (from Lisp) are only two famous examples.

2. the function which should be "folded" to achieve a special effect can be algorithmically deduced from the standard recursive definition of the required effect.

3. proving the correctness of some programs, which is usually made by induction, can be replaced with a simple check of two conditions.

All those aspects were studied in [3], so we are not repeating them, here..

2.2. The software platform: SWI-Prolog

The standard Prolog, as it was defined from the beginning by A. Colmerauer and Ph. Rusell , as it is described in classic manuals like [6] makes a clear distinction between functions and predicates. That is why we have switched to a more versatile, modern, version of Prolog, the SWI-Prolog, and also have a well documented manual [4]. It has some aditional

properties:

a) there exists a call predicate which can dynamically take a predicate and try to prove it.

b) predicates can be converted into what the authors of SWI-Prolog have called “arithmetic functions”.

Both those properties are interesting from the point of wiev of creating and using folds in logic programming.

SWI-Prolog is a freely available Prolog System which can be downloaded from the internet and is include in Linux distributions. It is based on some Dec Prolog libraries, being an extension of Dec Prolog. It is hosted on [8]. It is a product of the University of Amsterdam and its latest version of manual is signed by Jan Wielemaker. In the next pages, even when just Prolog is written, we mean SWI-Prolog.

2.3. Implementing metapredicates in SWI-Prolog

Metapredicates is a term which is describing predicates about predicates. Because in SWI-Prolog predicates may become functions and folds operator are accepting functions as arguments, makes sense to discus about metapredicates implementation in SWI-Prolog. We are using the term *metapredicates* in the same way that we have used the term *high order functions* in functional programming.

Before the discussion about folds implementation in Prolog, may be a good and simple enough challenge to implement the standard *map* operator from Lisp. In functional languages like Haskell, Lisp, etc, a *map* is a function which accepts a function of one argument and a list and applies the function to all the values from that list. After a bit of search in the [4] the solution is found:

```
/* How to define a metapredicate, episode I. */
/* using the SWI-Prolog predicate:  call()   */

/* Filename: metalogica3bv2.pl */
/* Here, mymap is doing the same thing as the maplist predefined predicate. */
/* He is receiving an arithmetic predicate and apply it over all the values from the
list. */
/* An other argument collects the results.*/

/*
Theory: "call" is an atom
```

```
system:call/6 is a built-in meta predicate defined in
        /usr/lib/swi-prolog/boot/init.pl:181
        Summary: ``Call with additional arguments"*/

/* Inspired by the SWI-Prolog 5.10 Reference Manual
 5.4. DEFINING A META-PREDICATE
 pg 187
 */


module(mymap, [mymap/3]).
meta_predicate mymap(2, ?, ?).

%% mymap(:Pred, +List1, ?List2)


mymap(_,[], []).                    /* Processing  [] we will get  an other [] */
mymap(Pred, [H0|T0], [H|T]) :-
                call(Pred, H0, H), /* Applying  Pred to H0 we can get  H, the head of
the result's list */
                        mymap(Pred,T0, T). /* The tail  T is obtaining applying  mymap
on the tail */
                                                /* of the argument, T0 .*/
```

The first *call* is applying the predicate *Pred* to the head H0 obtaining H, the head of the new computed list. The tail of this list is obtained applying *Pred,* using *mymap* to the tail of the first given list,T0.

We describe how it works. The above file can be loaded using: swpl -f <filename>. After that, you can launch interogations from the SWI-Prolog prompter. Let's rise the integer 2 at some different powers, for example:

```
?- mymap(pow(2),[1,3,4,11,10],Result).

Result = [2, 8, 16, 2048, 1024] ;

false.
```

After pressing the semi-column key “;”, the system is answering: “false”. This mean there is no other solution available. This is a normal behavior for a list processing function. Also, it means that Prolog actually did not need to backtrack in order to find more solutions.

2.4. Defining foldl as a metapredicate

Having the above experience with *mymap* we are ready to define a first fold, the *foldl* operator:

We are following the above model, the reader will probably see how.

```
/* How to define foldl as a metapredicate. Metapredicates, episode II  */

/*In the example below fold is a metapredicate which
   receive an arithmetic predicate (2 inputs, one output)
   and begin to compute  (..(V0 + V1) + ...)+ Vn,  where

   V0 and the list  [V1, ... Vn] are the last two arguments of foldl
  and the " +" operator can be any one selected by the user, from the
  list of arithmetic predicates, even user defined.
 */

/* (c) Dan Popa inspired by The SWI-Prolog 5.10 Reference Manual
, 5.4. DEFINING A META-PREDICATE, pp 187
 and  Graham Hutton's paper[3] concerning folds.
*/

module(foldl, [fold/4]).
meta_predicate foldl(0, 0, ?, ?).

%% foldl(:Pred, +V0, +List1, ?Rez)


foldl(Pred, V0, L1, R) :- fold_(V0, L1, R, Pred),!.  /* changing the order and cut */

fold_(_, [], [], _).
fold_(V0,[H0],Rezultat,Pred) :- call(Pred, V0,H0,Rezultat).   /* Base of induction. */
fold_(V0, [H0|T0], R, Pred) :- call(Pred, V0, H0, RezPart),
                                    fold_(RezPart, T0, R, Pred).
```

Let's see how it works. You should put a dot after the interrogation in order to avoid backtracking .

```
?- foldl(plus,0,[1,2,3],L).

L = 6 .



foldl(plus,0,[1,2,3],L), writeln(L).
```

```
6

L = 6 .


?- foldl(pow,2,[1,2,3],L).

L = 64 .


?- foldl(pow,2,[2,3,4],L).

L = 16777216 .


?- foldl(plus,0,[2,5,7],Rez).

Rez = 14.


?- foldl(pow,2,[2,2,2],Rez).

Rez = 256.


?-    foldl(pow,9,[9,9],Rez).

Rez =
196627050475552913618075908526912116283103450944214766927315415537966391196809.
```

3. Implementing foldr in Prolog

The main difference between *foldl* and *foldr* is the direction of scanning the list of values to be processed. Entering a reverse (which is reversing the list) in the program should transform foldl in a foldr. The new module of Prolog may looks like this:

```
/* How to define foldr as a metapredicate. Metapredicates, episode III .
*/
/* Filename: metalogica5foldr.pl */

/*In the example below foldr is a metapredicate which
  receive an arithmetic predicate (2 inputs, one output)
```

```
      and begin to compute  V1 x ...(... (Vn x V0)..),  where
      V0 and the list  [V1, ... Vn] are the last two arguments of foldl
     and the "x" operator can be any one selected by the user, from the
     list of arithmetic predicates, even user defined.

      In the end we will notice that cons then reverse are defined in terms of
foldl.
      And foldr can also be defined.
   */

   /* (c) Dan Popa inspired by The SWI-Prolog 5.10 Reference Manual
   , 5.4. DEFINING A META-PREDICATE, pp 187
    and  Graham Hutton's paper[3] concerning folds.
   */


   module(foldr, [foldr/4]).
   meta_predicate foldr(0, 0, ?, ?).


   %% foldr(:Goal, +V0, +List1, ?Rez)

   /* The first argument is the  binary arithmetic predicate, then
       the first value comes,
      folowed by the list of values to be processed.
      The final variable is used to collect the answer. */



   foldr(Goal, V0, L1, R) :- reverse(L1,LR), fold_(V0, LR, R, Goal).
/* reversing the list */
   fold_(V0,[H1],R,Goal)        :- call(Goal, H1,V0,R),!.
   fold_(V0, [Hn|Tn], R, Goal) :-     call(Goal, Hn, V0, RezPart),
                                          fold_(RezPart, Tn, R, Goal).
```

After the loading the program we can ask SWI-Prolog, for example, to evaluate the following folds (the use of the built in predicate *reverse* is shown, too):

```
   ? - foldr(plus,0,[1,2,3,4],R).

   R = 10.

   ?- foldr(plus,0,[1,2,3,4],R),writeln(R).

   10
```

```
R = 10.

?- reverse([1,2,3],[]).

false.

?- reverse([1,2,3],L).

L = [3, 2, 1].


?- explain(reverse).

"reverse" is an atom
lists:reverse/2 is a predicate defined in
        /usr/lib/swi-prolog/library/lists.pl:276
lists:reverse/4 is a predicate defined in
        /usr/lib/swi-prolog/library/lists.pl:279
        Referenced from 1-th clause of lists:reverse/2
        Referenced from 2-th clause of lists:reverse/4

true.
```

Remark 1: If the final *cut* (!) is missing, this implementation of *foldr* will finish by returning *false*.

```
?- foldr(plus,0,[1,2,3,4],R).

R = 10 ;

false.


?- foldr(plus,0,[1,2,3,4],R),writeln(R).

10

R = 10 ;

false.
```

Remark 2: In Haskell, the reverse function is itself a foldl.

```
reverse = foldl (\ xs x ->  x:xs) []
```

This leads us to the idea of defining *foldl*, then reverse, then *foldl*. In this way, *foldr* is defined using *foldr*, without the predefined function reverse.

Remark 3: In SWI-Prolog, the following predicate may be the considered the equivalent of the *cons* operator:

cons (A,As,R) :- R is [A|AS].

So, the cons (:) from the above definition of reverse can be implemented in SWI-Prolog, too, as a predicate and, if needed, can be appealed by using the *call* predicate.

4. THE FOLDR OPERATOR DEFINED BY USING ONLY FOLDL

In[3] the *reverse* function is defined, as is noted in the above Remark 2, also as a fold. That means we can use this kind of definition inside of foldl, before the processing of the list, in order to transform the foldl in a foldr. We begin by defining a sort of cons operator, but simpler that those from Remark 3. Let rename it as *lambda*. (This new name should work if the above *cons* is already included in our source.)

lambda (XS,X,[XS])

Now, we can use this, from SWI-Prolog, in order to *reverse* lists, with the help of a *foldl*. Here is an example:

```
? foldl (lambda,[ ], [1,2,3], Rez).

Rez=[3,2,1]
```

As a consequence, we can define a little different version of foldr by simply changing the standard *reverse* predicate with the above *foldl*. This makes our folds a bit more independent of the standard libraries.

```
foldr(Pred, V0, L1, R) :- foldl(lambda, [], L1, Reversed)
                          fold_(V0, Reversed, R, Pred),!.
```

Where *fold_* and *foldl* are the same as above. Now all the results from [3] can be also used by SWI-Prolog programmers. This kind of fold based in Prolog is not appearing in any of the manuals of Prolog, [1], [2], [4], [6], [7] (see a selection of them in the references).

5. THE USE OF FOLDR

In order to test one of our implementations of the *foldr* operator in SWI-Prolog we have reconsider the examples of *foldr* use from[3]. As you can se below, all the folds were successfully computed. The fact that a lot of

common used functions are in fact folds was clearly stated in[3]. The reader can easy recognize: the factorial n!, catenation, etc.

```
/* Folds in Graham Hutton's style */

/* DP */
/* Filename: metalogica8foldrGH.pl */

arithmetic_function(foldr/3).  /* Not absolutely necessary, but may help. */
arithmetic_function(mult/3).
arithmetic_function(and/3).
arithmetic_function(or/3).
arithmetic_function(cons/3).
arithmetic_function(f1/3).
arithmetic_function((++)/3).
arithmetic_function(f2/3).


foldr(_,V,    [],V2 ) :-  V2=V,!.

foldr(F,V,[X|XS],Rez) :-  foldr(F,V,XS,RezPart),
                          call(F,X,RezPart,Rez).


mult(A,B,C) :- C is A*B.
and(A,B,true) :- A, B.
or(A,_,true) :- A,!.
or(_,B,true) :- B,!.
cons(A,B,C) :- C = [A|B].
f1(_,B,C) :- plus(1,B,C).              /* Te reader may also try: C is 1+B */
++(A,B,C) :-  foldr(cons, B,A,C).
f2(A,B,C) :- ++(B,[A],C).
```

And let's ask SWI-Prolog to compute some common functions, sum or product, && and disjunctions (on list of booleans), factorial, lists's catenation , or even fold some user-defined functions like f1 and f2.

```
?- foldr(plus,0,[1,2,3],Rez).

Rez = 6.

?- foldr(mult,1,[1,2,3,4,5],NFact).

NFact = 120.

?- and(true,true,X).
```

```
X = true.

?- and(true,false,X).

false.

?- and(false,true,X).

false.

?- and(false,false,X).

false.

?- foldr(and,true,[true,true,true],Rez).

Rez = true.


?- foldr(and,true,[true,false,true],Rez).

false.

?- or(true,true,X).

X = true.

?- or(true,false,X).

X = true.

?- or(false,false,X).

false.

?- or(false,true,X).

X = true.


?- foldr(or,false,[true,true,true],Rez).

Rez = true.


?- foldr(or,false,[true,false,true],Rez).

Rez = true.
```

```
?- foldr(or,false,[true,false,false],Rez).

false.

?- foldr(or,false,[false,false,false],Rez).

false.

?- foldr(cons,[2,3,4],[1,2],Rez).

Rez = [1, 2, 2, 3, 4].


?- foldr(f1,0,[1,3,5,9,1],Rez).

Rez = 5.
?- ++([1,2,3],[4,5,6],Rez).

Rez = [1, 2, 3, 4, 5, 6].


?- foldr(f2,[],[2,1,1,2,6,9],Rez).

Rez = [9, 6, 2, 1, 1, 2].
```

6. HISTORY AND PRESENT

We have taught our students about foldl, foldr and fold based programming in Prolog since 2012, the year when the author had to create a mixed course of Functional and Logic Programming. Nowadays, the latest SWI-Prolog Reference manual (6.6.2) is actually including a reference to an implementation of only one operator, the foldl, in the section A.2. LIBRARY (apply): Apply predicates on a list pp 332-333:

```
foldl(:Goal, +List, +V0, -V)
foldl(:Goal, +List1, +List2, +V0, -V)
foldl(:Goal, +List1, +List2, +List3, +V0, -V)
foldl(:Goal, +List1, +List2, +List3, +List4, +V0, -V)
```

The library is including also an other well known Haskell function scanl reimplemented for SWI-Prolog but no implementation of foldr, yet.

REFERENCES

[1] P. Blackburn, J. Bos, K. Striegnitz - **Learn Prolog Now!,** http://www.learnprolognow.org/ (on line resource)

[2] D. Diaz, **GNU PROLOG, A Native Prolog Compiler with Constraint Solving over Finite Domains**,Edition 1.31, for GNU Prolog version 1.4.1, June 6, 2012, http://people.sju.edu/~jhodgson/clp/manualgp.pdf , (on line resource)

[3] G. Hutton, - **A tutorial on the universality and expressiveness of fold**, Journal of Functional Programming, vol. 9, Issue 04, July 1999, 355-372 and J.F.P 1(1)-000 January 1993 - Cambridge University, http://www.cs.nott.ac.uk/~gmh/bib.html#fold (on line resource)

[4] J. Wielemaker - **SWI-Prolog 5.10 Reference Manual Updated for version 5.10.0**, April 2010 -University of Amsterdam, The Netherlands, (on line resource)

[5] J. Wielemaker - **SWI-Prolog 6.6.2 Reference Manual Updated for version 6.6.2**, March 2014, (on line resource)

[6] A.L. Johanson, A. Eriksson-Granskog, A. Edman – **Prolog versus you – An introduction to Logic Programming** – Springer Verlag, 1989. ISBN 3-540-17577-6, ISBN 0-387-17577-6

[7] Lu James, Mead Jerud J. -Prolog - **A Tutorial Introduction** - Computer Science Department, Bucknell University (on line resource)

[8] The SWI-Prolog website http://www.swi-prolog.org

"Vasile Alecsandri" University of Bacău
Faculty of Sciences
Department of Mathematics, Informatics and Education Sciences
157 Calea Mărăşeşti, Bacău, 600115, ROMANIA
e-mail: popavdan@yahoo.com